Principles for secure design

Some of the slides and content are from Mike Hicks’ Coursera course
Making secure software

• **Flawed approach**: Design and build software, and *ignore security at first*
  • Add security once the functional requirements are satisfied

• **Better approach**: *Build security in* from the start
  • Incorporate security-minded thinking into all phases of the development process
Development process

Four common phases of development

- Requirements
- Design
- Implementation
- Testing/assurance

Security activities apply to all phases

- Security Requirements
- Abuse Cases
- Architectural Risk Analysis
- Security-oriented Design
- Code Review (with tools)
- Risk-based Security Tests
- Penetration Testing
Four common phases of development

- Requirements
- Design
- Implementation
- Testing/assurance

Security activities apply to all phases

We’ve been talking about these
Development process

Four common phases of development

- Requirements
- Design
- Implementation
- Testing/assurance

This class is about these

We’ve been talking about these

Security activities apply to all phases

- Security Requirements
- Abuse Cases
- Architectural Risk Analysis
- Security-oriented Design
- Code Review (with tools)
- Risk-based Security Tests
- Penetration Testing
Designing secure systems

- **Model** your threats

- Define your **security requirements**
  - What distinguishes a security requirement from a typical “software feature”?

- Apply good security **design principles**
Threat Modeling
Threat Model

• The **threat model** makes explicit the adversary’s **assumed powers**
  • Consequence: The threat model must match reality, otherwise the risk analysis of the system will be wrong

• The threat model is **critically important**
  • If you are not explicit about what the attacker can do, how can you assess whether your design will repel that attacker?
Threat Model

- The threat model makes explicit the adversary’s assumed powers
  - Consequence: The threat model must match reality, otherwise the risk analysis of the system will be wrong

- The threat model is critically important
  - If you are not explicit about what the attacker can do, how can you assess whether your design will repel that attacker?

“This system is secure” means nothing in the absence of a threat model
A few different network threat models
A few different network threat models
A few different network threat models

Malicious user

Snooping

Co-located user
A few different network threat models

- Malicious user
- Snooping
- Co-located user
- Compromised server
Threat-driven Design

• Different threat models will elicit different responses

• Only malicious users: implies message traffic is safe
  • No need to encrypt communications
  • This is what telnet remote login software assumed

• Snooping attackers: means message traffic is visible
  • So use encrypted wifi (link layer), encrypted network layer (IPsec), or encrypted application layer (SSL)
    - Which is most appropriate for your system?

• Co-located attacker: can access local files, memory
  • Cannot store unencrypted secrets, like passwords
  • Likewise with a compromised server

More on these when we get to networking

In fact, even encrypting them might not suffice! (More later)
Threat-driven Design

• Different threat models will elicit different responses

• Only malicious users: implies message traffic is safe
  • No need to encrypt communications
  • This is what telnet remote login software assumed

• Snooping attackers: means message traffic is visible
  • So use encrypted wifi (link layer), encrypted network layer (IPsec), or encrypted application layer (SSL)
    - Which is most appropriate for your system?

• Co-located attacker: can access local files, memory
  • Cannot store unencrypted secrets, like passwords
  • Likewise with a compromised server

More on these when we get to networking
In fact, even encrypting them might not suffice! (More later)
Bad Model = Bad Security

• Any assumptions you make in your model are potential holes that the adversary can exploit
Bad Model = Bad Security

• Any assumptions you make in your model are potential holes that the adversary can exploit

• E.g.: Assuming no snooping users no longer valid
  • Prevalence of wi-fi networks in most deployments
Bad Model = Bad Security

- Any **assumptions** you make in your model are potential **holes that the adversary can exploit**

- E.g.: Assuming no snooping users **no longer valid**
  - **Prevalence of wi-fi networks in most deployments**

- Other mistaken assumptions
  - **Assumption**: Encrypted traffic carries no information
Bad Model = Bad Security

- Any **assumptions** you make in your model are potential **holes that the adversary can exploit**

- E.g.: Assuming no snooping users **no longer valid**
  - *Prevalence of wi-fi networks in most deployments*

- Other mistaken assumptions
  - **Assumption**: Encrypted traffic carries no information
    - Not true! By analyzing the size and distribution of messages, you can infer application state
  - **Assumption**: Timing channels carry little information
    - Not true! Timing measurements of previous RSA implementations could be used eventually reveal a remote SSL secret key
Bad Model = Bad Security

**Assumption:** Encrypted traffic carries no information

**Skype** encrypts its packets, so we’re not revealing anything, right?

Language Identification of Encrypted VoIP Traffic:
*Alejandra y Roberto or Alice and Bob?*

Charles V. Wright    Lucas Ballard    Fabian Monroe    Gerald M. Masson

But Skype varies its packet sizes…

![Figure 2: Unigram frequencies of bit rates for English, Brazilian Portuguese, German and Hungarian](image)
Bad Model = Bad Security

**Assumption:** Encrypted traffic carries no information

Skype encrypts its packets, so we’re not revealing anything, right?

But Skype varies its packet sizes…

…and different languages have different word/unigram lengths…

---

**Language Identification of Encrypted VoIP Traffic:**

*Alejandra y Roberto or Alice and Bob?*

*Charles V. Wright | Lucas Ballard | Fabian Monrose | Gerald M. Masson*

---

![Unigram frequencies for Skype bit rates](image)

*Figure 2: Unigram frequencies of bit rates for English, Brazilian Portuguese, German and Hungarian*
Bad Model = Bad Security

Assumption: Encrypted traffic carries no information

Skype encrypts its packets, so we’re not revealing anything, right?

But Skype varies its packet sizes...

...and different languages have different word/unigram lengths...

...so you can infer what language two people are speaking based on packet sizes!

Figure 2: Unigram frequencies of bit rates for English, Brazilian Portuguese, German and Hungarian
Finding a good model

- **Compare against similar systems**
  - What attacks does their design contend with?

- **Understand past attacks and attack patterns**
  - How do they apply to your system?

- **Challenge assumptions in your design**
  - What happens if an assumption is untrue?
    - What would a breach potentially cost you?
  - How hard would it be to get rid of an assumption, allowing for a stronger adversary?
    - What would that development cost?
You have your threat model.

Now let’s define what we need to defend against.

Security Requirements
Security Requirements

- **Software requirements** typically about **what** the software should do

- We also want to have **security requirements**
  - **Security-related goals** (or **policies**)
    - **Example**: One user’s bank account balance should not be learned by, or modified by, another user, unless authorized
  - **Required mechanisms** for enforcing them
    - **Example**:
      1. Users identify themselves using passwords,
      2. Passwords must be “strong,” and
      3. The password database is only accessible to login program.
Typical *Kinds* of Requirements

- **Policies**
  - **Confidentiality** (and Privacy and Anonymity)
  - Integrity
  - Availability

- **Supporting mechanisms**
  - Authentication
  - Authorization
  - Audit-ability
  - Encryption
Supporting mechanisms

These relate identities ("principals") to actions

Authentication          Authorization          Audit-ability
Supporting mechanisms

These relate identities ("principals") to actions

**Authentication**
How can a system tell *who a user is*

**Authorization**

**Audit-ability**
Supporting mechanisms

These relate identities (“principals”) to actions

**Authentication**
How can a system tell *who a user is*

- What we know
- What we have
- What we *are*

>1 of the above = *Multi-factor authentication*

**Authorization**

**Audit-ability**
Supporting mechanisms

These relate identities ("principals") to actions

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Authorization</th>
<th>Audit-ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>How can a system tell <em>who a user is</em></td>
<td>How can a system tell <em>what a user is allowed to do</em></td>
<td></td>
</tr>
</tbody>
</table>

What we know
What we have
What we *are*
>1 of the above = *Multi-factor authentication*
Supporting mechanisms

These relate identities ("principals") to actions

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Authorization</th>
<th>Audit-ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>How can a system</td>
<td>How can a system</td>
<td>Access control policies</td>
</tr>
<tr>
<td>tell who a user is</td>
<td>tell what a user is</td>
<td>(defines)</td>
</tr>
<tr>
<td></td>
<td>allowed to do</td>
<td>+</td>
</tr>
<tr>
<td>What we know</td>
<td></td>
<td>Mediator</td>
</tr>
<tr>
<td>What we have</td>
<td></td>
<td>(checks)</td>
</tr>
<tr>
<td>What we are</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;1 of the above =</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-factor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>authentication</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Supporting mechanisms

These relate identities ("principals") to actions

<table>
<thead>
<tr>
<th><strong>Authentication</strong></th>
<th><strong>Authorization</strong></th>
<th><strong>Audit-ability</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>How can a system tell <em>who a user is</em></td>
<td>How can a system tell <em>what a user is allowed to do</em></td>
<td>How can a system tell <em>what a user did</em></td>
</tr>
</tbody>
</table>

What we know
What we have
What we *are*

>1 of the above = *Multi-factor authentication*

Access control policies
(defines)

+ *Mediator*
(changes)
# Supporting mechanisms

These relate identities ("principals") to actions.

<table>
<thead>
<tr>
<th>Authentication</th>
<th>Authorization</th>
<th>Audit-ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>How can a system tell <em>who a user is</em></td>
<td>How can a system tell <em>what a user is allowed to do</em></td>
<td>How can a system tell <em>what a user did</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What we know</th>
<th>Access control policies (defines)</th>
<th>Retain enough info to determine the circumstances of a breach</th>
</tr>
</thead>
<tbody>
<tr>
<td>What we have</td>
<td>+ <em>Mediator</em> (checks)</td>
<td></td>
</tr>
<tr>
<td>What we <em>are</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;1 of the above</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Multi-factor authentication*
Defining Security Requirements

• Many processes for deciding security requirements

• Example: **General policy concerns**
  • Due to **regulations**/standards (HIPAA, SOX, etc.)
  • Due **organizational values** (e.g., valuing privacy)

• Example: **Policy arising from threat modeling**
  • Which **attacks** cause the **greatest concern**?
    - Who are the likely adversaries and what are their goals and methods?
  • Which **attacks** have **already occurred**?
    - Within the organization, or elsewhere on related systems?
Abuse Cases

• Abuse cases illustrate security requirements

• Where use cases describe what a system should do, abuse cases describe what it should not do

• Example use case: The system allows bank managers to modify an account’s interest rate

• Example abuse case: A user is able to spoof being a manager and thereby change the interest rate on an account
Defining Abuse Cases

• Construct cases in which an adversary’s exercise of power could violate a security requirement
  • Based on the threat model
  • What might occur if a security measure was removed?

• **Example**: Co-located attacker steals password file and learns all user passwords
  • Possible if password file is not encrypted

• **Example**: Snooping attacker replays a captured message, effecting a bank withdrawal
  • Possible if messages are have no nonce (a small amount of uniqueness/randomness - like the time of day or sequence number)
Security design principles
Design Defects = Flaws

- Recall that software defects consist of both flaws and bugs
  - **Flaws** are problems in the design
  - **Bugs** are problems in the implementation

- We avoid flaws during the design phase

- According to Gary McGraw, **50% of security problems are flaws**
  - So this phase is very important
Categories of Principles
Categories of Principles

• Prevention
  • **Goal**: Eliminate software defects entirely
  • **Example**: Heartbleed bug would have been prevented by using a type-safe language, like Java
Categories of Principles

• **Prevention**
  • **Goal**: Eliminate software defects entirely
  • **Example**: Heartbleed bug would have been prevented by using a type-safe language, like Java

• **Mitigation**
  • **Goal**: Reduce the harm from exploitation of unknown defects
Categories of Principles

• **Prevention**
  • **Goal**: Eliminate software defects entirely
  • **Example**: Heartbleed bug would have been prevented by using a type-safe language, like Java

• **Mitigation**
  • **Goal**: Reduce the harm from exploitation of unknown defects
  • **Example**: Run each browser tab in a separate process, so exploitation of one tab does not yield access to data in another

• **Detection** (and **Recovery**)
  • **Goal**: Identify and understand an attack (and undo damage)
  • **Example**: Monitoring (e.g., expected invariants), snapshotting
Principles for building secure systems

General rules of thumb that, when neglected, result in design flaws

- Security is economics
- Principle of least privilege
- Use fail-safe defaults
- Use separation of responsibility
- Defend in depth
- Account for human factors
- Ensure complete mediation
- Kerkhoff’s principle

- Accept that threat models change
- If you can’t prevent, detect
- Design security from the ground up
- Prefer conservative designs
- Proactively study attacks
“Security is economics”

You can’t afford to secure against *everything*, so what *do* you defend against? Answer: That which has the greatest “return on investment”

**THERE ARE NO SECURE SYSTEMS, ONLY DEGREES OF INSECURITY**

- In practice, need to **resist a certain level of attack**
  - Example: Safes come with security level ratings
  - “Safe against safecracking tools & 30 min time limit”

- Corollary: Focus energy & time on **weakest link**

- Corollary: Attackers follow the *path of least resistance*
“Principle of least privilege”

Give a program the access it legitimately needs to do its job. NOTHING MORE

• This doesn’t necessarily reduce probability of failure
• Reduces the EXPECTED COST

• **Example**: Unix does a BAD JOB:
  • Every program gets all the privileges of the user who invoked it
  • `vim` as root: it can do anything -- should just get access to file

• **Example**: Windows JUST AS BAD, MAYBE WORSE
  • Many users run as Administrator,
  • Many tools require running as Administrator
“Use fail-safe defaults”

Things are going to break. Break safely.

- **Default-deny policies**
  - Start by denying all access
  - Then allow only that which has been explicitly permitted

- **Crash =&gt; fail to secure behavior**
  - Example: firewalls explicitly decide to forward
  - Failure =&gt; packets don’t get through
“Use separation of responsibility”

Split up privilege so no **one** person or program has total power.

- **Example**: US government
  - Checks and balances among different branches

- **Example**: Movie theater
  - One employee sells tickets, another tears them
  - Tickets go into lockbox

- **Example**: Nuclear weapons…
Use separation of responsibility
“Defend in depth”

Use multiple, redundant protections

• Only in the event that *all of them* have been breached should security be endangered.

• **Example**: Multi-factor authentication:
  • Some combination of password, image selection, USB dongle, fingerprint, iris scanner,… (more on these later)

• **Example**: “You can recognize a security guru who is particularly cautious if you see someone wearing both….”
...a belt and suspenders
Defense in depth

...a belt and suspenders
“Ensure complete mediation”

Make sure your reference monitor sees *every* access to *every* object

- Any **access control system** has some resource it needs to enforce
  - Who is allowed to access a file
  - Who is allowed to post to a message board...

- **Reference Monitor**: The piece of code that checks for permission to access a resource
Ensure complete mediation
“Account for human factors”

(1) “Psychological acceptability”: Users must buy into the security model

• The security of your system ultimately lies in the hands of those who use it.

• If they don’t believe in the system or the cost it takes to secure it, then they won’t do it.

• **Example**: “All passwords must have 15 characters, 3 numbers, 6 hieroglyphics, …”
Log in to your message center.
Invalid log in or server error. Please try again.
Forgot your password?

Log In Address: xazoo@netnet.com
example: john234@jumbowidgetsco.com

Password: ********
note: password is case-sensitive

Remember my Address and Password (what is this?)

LOG IN
Account for human factors ("psychological acceptability")

1. Users must buy into the security
“Account for human factors”

(2) The security system must be usable

- The security of your system ultimately lies in the hands of those who use it.

- If it is too hard to act in a secure fashion, then they won’t do it.

- **Example**: Popup dialogs
Account for human factors
(2) The security system must be usable
Account for human factors
(2) The security system must be usable
Account for human factors

(2) The security system must be usable
Account for human factors

(2) The security system must be usable
“Account for human factors”

(2) The security system must be usable

- The security of your system ultimately lies in the hands of those who use it.
- If it is too hard to act in a secure fashion, then they won’t do it.
- **Example**: Popup dialogs
“Kerkhoff’s principle”

Don’t rely on security through obscurity

• Originally defined in the context of crypto systems (encryption, decryption, digital signatures, etc.):

• Crypto systems should remain secure even when an attacker knows all of the internal details
  • It is easier to change a compromised key than to update all code and algorithms

• The best security is the light of day
Kerkhoff’s principle??
Kerkhoff’s principle!
Principles for building secure systems

Know these well:
• Security is economics
• Principle of least privilege
• Use fail-safe defaults
• Use separation of responsibility
• Defend in depth
• Account for human factors
• Ensure complete mediation
• Kerkhoff’s principle

Self-explanatory:
• Accept that threat models change; adapt your designs over time
• If you can’t prevent, detect
• Design security from the ground up
• Prefer conservative designs
• Proactively study attacks
SANDBOXES

Execution environment that restricts what an application running in it can do

**NaCl’s restrictions**
- Takes arbitrary x86, runs it in a sandbox in a browser
- Restrict applications to using a narrow API
- Data integrity: No reads/writes outside of sandbox
- No unsafe instructions
- CFI

**Chromium’s restrictions**
- Runs each webpage’s rendering engine in a sandbox
- Restrict rendering engines to a narrow “kernel” API
- Data integrity: No reads/writes outside of sandbox (incl. the desktop and clipboard)
What have I done to deserve this?
Even the untrusted code needs input and output

The goal of the sandbox is to constrain what the untrusted program can execute, what data it can access, what system calls it can make, etc.
Example sandboxing mechanism: SecComp

- Linux system call enabled since 2.6.12 (2005)
  - Affected process can subsequently **only perform read, write, exit, and sigreturn system calls**
  - No support for **open** call: Can only use already-open file descriptors
  - **Isolates a process by limiting possible interactions**

- Follow-on work produced **seccomp-bpf**
  - **Limit process to policy-specific set of system calls**, subject to a policy handled by the kernel
    - Policy akin to *Berkeley Packet Filters (BPF)*
  - **Used by Chrome, OpenSSH, vsftpd, and others**
Idea: Isolate Flash Player
Idea: Isolate Flash Player

• Receive .swf code, save it
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call `fork` to create a new process
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call `fork` to create a new process
- In the new process, open the file
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call **fork** to create a new process
- In the new process, open the file
- Call **exec** to run Flash player
Idea: Isolate Flash Player

- Receive .swf code, save it
- Call `fork` to create a new process
- In the new process, open the file
- Call `exec` to run Flash player
- Call `seccomp-bpf` to compartmentalize
Sandboxing as a design principle

- It’s not just 3rd-party code that should be sandboxed: sandbox your own code, too!
- Break up your program into modules that separate responsibilities (what you should be doing anyway)
- Give each module the least privileges it needs to do its job
- Use the sandbox to enforce what exactly a given module can/can’t do

- 3rd party binaries (NaCl)
- Webpages (Chromium)
- Modules of your own code:
  Mitigate the impact of the inevitability that your code has an exploitable bug
Case study: VSFTPD
Very Secure FTPD

- **FTP**: File Transfer Protocol
  - More popular before the rise of HTTP, but still in use
  - 90's and 00's: **FTP daemon compromises were frequent and costly**, e.g., in Wu-FTPD, ProFTPd, …

- Very **thoughtful design** aimed to **prevent** and **mitigate** security defects

- But also to **achieve good performance**
  - Written in C

- Written and maintained by Chris Evans since 2002
  - **No security breaches that I know of**

https://security.appspot.com/vsftpd.html
VSFTPD Threat model

- Clients untrusted, until authenticated

- Once authenticated, limited trust:
  - According to user’s file access control policy
  - For the files being served FTP (and not others)

- Possible attack goals
  - Steal or corrupt resources (e.g., files, malware)
  - Remote code injection

- Circumstances:
  - Client attacks server
  - Client attacks another client
struct mystr
{
    char* PRIVATE_HANDS_OFF_p_buf;
    unsigned int PRIVATE_HANDS_OFF_len;
    unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
Defense: Secure Strings

```c
struct mystr
{
    char* PRIVATE_HANDS_OFF_p_buf;
    unsigned int PRIVATE_HANDS_OFF_len;
    unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Normal (zero-terminated) C string
struct mystr
{
    char* PRIVATE_HANDS_OFF_p_buf;
    unsigned int PRIVATE_HANDS_OFF_len;
    unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};

Normal (zero-terminated) C string

The actual length (i.e., strlen(PRIVATE_HANDS_OFF_p_buf))
Defense: Secure Strings

```c
struct mystr
{
    char* PRIVATE_HANDS_OFF_p_buf;
    unsigned int PRIVATE_HANDS_OFF_len;
    unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Normal (zero-terminated) C string

The actual length (i.e., `strlen(PRIVATE_HANDS_OFF_p_buf)`) 

Size of buffer returned by `malloc`
Defense: Secure Strings

```c
struct mystr {
  char* PRIVATE_HANDS_OFF_p_buf;
  unsigned int PRIVATE_HANDS_OFF_len;
  unsigned int PRIVATE_HANDS_OFF_alloc_bytes;
};
```

Normal (zero-terminated) C string
The actual length (i.e., `strlen(PRIVATE_HANDS_OFF_p_buf)`) Size of buffer returned by `malloc`
```c
void private_str_alloc_memchunk(struct mystr* p_str, const char* p_src, unsigned int len)
{
    ...
}

void str_copy(struct mystr* p_dest, const struct mystr* p_src)
{
    private_str_alloc_memchunk(p_dest, p_src->p_buf, p_src->len);
}

struct mystr
{
    char* p_buf;
    unsigned int len;
    unsigned int alloc_bytes;
};

replace uses of char* with struct mystr*
and uses of strcpy with str_copy
```
void
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src, unsigned int len)
{
    /* Make sure this will fit in the buffer */
    unsigned int buf_needed;
    if (len + 1 < len)
    {
        bug("integer overflow");
    }
    buf_needed = len + 1;
    if (buf_needed > p_str->alloc_bytes)
    {
        str_free(p_str);
        s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
        p_str->alloc_bytes = buf_needed;
    }
    vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
    p_str->p_buf[len] = '\0';
    p_str->len = len;
}

struct mystr
{
    char* p_buf;
    unsigned int len;
    unsigned int alloc_bytes;
};

Copy in at most len bytes from p_src into p_str
void private_str_alloc_memchunk(struct mystr* p_str, const char* p_src, unsigned int len)
{
    /* Make sure this will fit in the buffer */
    unsigned int buf_needed;
    if (len + 1 < len)
    {
        bug("integer overflow");
    }
    buf_needed = len + 1;
    if (buf_needed > p_str->alloc_bytes)
    {
        str_free(p_str);
        s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
        p_str->alloc_bytes = buf_needed;
    }
    vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
    p_str->p_buf[len] = '\0';
    p_str->len = len;
}
void 
private_str_alloc_memchunk(struct mystr* p_str, const char* p_src, 
    unsigned int len) 
{
    /* Make sure this will fit in the buffer */
    unsigned int buf_needed;
    if (len + 1 < len)
    {
        bug("integer overflow");
    }
    buf_needed = len + 1;
    if (buf_needed > p_str->alloc_bytes)
    {
        str_free(p_str);
        s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
        p_str->alloc_bytes = buf_needed;
    }
    vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
    p_str->p_buf[len] = '\0';
    p_str->len = len;
}

struct mystr
{
    char* p_buf;
    unsigned int len;
    unsigned int alloc_bytes;
};

**consider NUL terminator when computing space**

**allocate space, if needed**

**Copy in at most len bytes from p_src into p_str**
void private_str_alloc_memchunk(struct mystr* p_str, const char* p_src, unsigned int len)
{
    /* Make sure this will fit in the buffer */
    unsigned int buf_needed;
    if (len + 1 < len)
    {
        bug("integer overflow");
    }
    buf_needed = len + 1;
    if (buf_needed > p_str->alloc_bytes)
    {
        str_free(p_str);
        s_setbuf(p_str, vsf_sysutil_malloc(buf_needed));
        p_str->alloc_bytes = buf_needed;
    }
    vsf_sysutil_memcpy(p_str->p_buf, p_src, len);
    p_str->p_buf[len] = '\0';
    p_str->len = len;
}

struct mystr
{
    char* p_buf;
    unsigned int len;
    unsigned int alloc_bytes;
};
Defense: Secure Stdcalls

- Common problem: error handling
Defense: Secure Stdcalls

- Common problem: **error handling**
- Libraries **assume** that **arguments are well-formed**
Defense: Secure Stdcalls

- Common problem: **error handling**
  - Libraries **assume** that arguments are well-formed
  - Clients **assume** that library calls always succeed
Defense: Secure Stdcalls

• Common problem: **error handling**
  • Libraries **assume** that arguments are well-formed
  • Clients **assume** that library **calls always succeed**

• Example: `malloc()`
Defense: Secure Stdcalls

• Common problem: **error handling**
  • Libraries **assume** that **arguments are well-formed**
  • Clients **assume** that library **calls always succeed**

• Example: **malloc()**
  • What if **argument is non-positive?**
    - We saw earlier that integer overflows can induce this behavior
    - Leads to buffer overruns
Defense: Secure Stdcalls

• Common problem: error handling
  • Libraries assume that arguments are well-formed
  • Clients assume that library calls always succeed

• Example: malloc()
  • What if argument is non-positive?
    - We saw earlier that integer overflows can induce this behavior
    - Leads to buffer overruns
  • What if returned value is NULL?
    - Oftentimes, a de-reference means a crash
    - On platforms without memory protection, a dereference can cause corruption
void*
vsf_sysutil_malloc(unsigned int size)
{
    void* p_ret;
    /* Paranoia - what if we got an integer overflow/underflow? */
    if (size == 0 || size > INT_MAX)
    {
        bug("zero or big size in vsf_sysutil_malloc");
    }
    p_ret = malloc(size);
    if (p_ret == NULL)
    {
        die("malloc");
    }
    return p_ret;
}
void*
vsf_sysutil_malloc(unsigned int size)
{
  void* p_ret;
  /* Paranoia - what if we got an integer overflow/underflow? */
  if (size == 0 || size > INT_MAX)
  {
    bug("zero or big size in vsf_sysutil_malloc");
  }
  p_ret = malloc(size);
  if (p_ret == NULL)
  {
    die("malloc");
  }
  return p_ret;
}
Defense: Minimal Privilege
Defense: Minimal Privilege

• **Untrusted input** always handled by **non-root process**
  • Uses IPC to delegate high-privilege actions
    - Very little code runs as **root**
Defense: Minimal Privilege

• **Untrusted input** always handled by **non-root process**
  • Uses IPC to delegate high-privilege actions
    - Very little code runs as root

• **Reduce privileges** as much as possible
  • Run as particular (unprivileged) user
    - File system access control enforced by OS
  • Use capabilities and/or SecComp on Linux
    - Reduces the system calls a process can make
Defense: Minimal Privilege

- **Untrusted input** always handled by **non-root process**
  - Uses IPC to delegate high-privilege actions
    - Very little code runs as root

- **Reduce privileges** as much as possible
  - Run as particular (unprivileged) user
    - File system access control enforced by OS
  - Use capabilities and/or SecComp on Linux
    - Reduces the system calls a process can make

- **chroot to hide all directories** but the current one
  - Keeps visible only those files served by FTP
Defense: Minimal Privilege

- **Untrusted input** always handled by **non-root process**
  - Uses IPC to delegate high-privilege actions
    - Very little code runs as `root`

- **Reduce privileges** as much as possible
  - Run as particular (unprivileged) user
    - File system access control enforced by OS
  - Use capabilities and/or SecComp on Linux
    - Reduces the system calls a process can make

- **chroot to hide all directories** but the current one
  - Keeps visible only those files served by FTP
Defense: Minimal Privilege

- **Untrusted input** always handled by **non-root process**
  - Uses IPC to delegate high-privilege actions
    - Very little code runs as **root**

- **Reduce privileges** as much as possible
  - Run as particular (unprivileged) user
    - File system access control enforced by OS
  - Use capabilities and/or SecComp on Linux
    - Reduces the system calls a process can make

- **chroot** to hide all directories but the current one
  - Keeps visible only those files served by FTP
Connection Establishment

correction
server

client
Connection Establishment

connection

server

TCP conn request

client
Connection Establishment

connection
server

command
processor

client
Connection Establishment

- connection server
- command processor
- login reader
- client
Connection Establishment

connection server

command processor

login reader

client

U+P

OK

OK

USER, PASS
Connection Establishment

- connection server
- command processor
- command reader/executor
- client
Performing Commands

connection server
command processor
command reader/executor
client
Performing Commands

connection server

command processor

command reader/executor

client

OK

CHDIR
Performing Commands

connection server → command processor
command reader/executor → client

CHOWN → OK

CHOWN → OK
Logging out

- connection server
- command processor
- command reader/executor
- client
Logging out
Attack: Login

- connection server
- command processor
- login reader
- client
Attack: Login

connection server

command processor

login reader

client

ATTACK
Attack: Login

- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
Attack: Login

- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code
Attack: Login

- **Login reader white-lists input**
  - And allowed input very limited
  - Limits attack surface
- **Login reader has limited privilege**
  - Not root; authentication in separate process
  - Mutes capabilities of injected code
- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input
Attack: Commands

- connection server
- command processor
- command reader/executor
- client
Attack: Commands

- connection server
- command processor
- command reader/executor
- client
Attack: Commands

- **Command reader sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege
Attack: Commands

- **Command reader sandboxed**
  - Not root
  - Handles most commands
  - Except few requiring privilege

- **Comm. proc. only talks to reader**
  - And, again, white-lists its limited input
Attack: Cross-session
Attack: Cross-session

connection server

command reader/executor

client 1

client 2
Attack: Cross-session

classifier
connection server
classifier
command reader/executor
classifier
command reader/executor
client 1
client 2
Attack: Cross-session

connection server

command processor

command reader/executor

client 1

command processor

command reader/executor

client 2
Attack: Cross-session
Attack: Cross-session

• Each session isolated
• Only can talk to one client
Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Presentation of vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it’s actually quite hard to do, but vsftpd manages it in the name of being secure. I’m unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. All OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

TCB: KISS

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrust. Here are examples requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

TCB: KISS

Presenting vsftpd's secure design
==================================

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Presenting vsftpd's secure design
================================

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request: The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown’ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it’s actually quite hard to do, but vsftpd manages it in the name of being secure. I’m unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Presenting vsftpd's secure design

==================================================================

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details
     are correct does the privileged parent launch a new child with the appropriate
     user credentials.
   - chown() request. The child may request a recently uploaded file gets
     chown'ed() to root for security purposes. The parent is careful to only allow
     chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to
     emit data connections from port 20. This requires privilege. The privileged
     parent process creates the privileged socket and passes it to child over
     the socket.

4) This same privileged parent process makes use of capabilities and chroot(),
to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
 privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
 protocol parsing is performed in a chroot() jail, running under an unprivileged
 user. This means both pre-authenticated and post-authenticated OpenSSL protocol
 parsing; it's actually quite hard to do, but vsftpd manages it in the name of
 being secure. I'm unaware of any other FTP server which supports both SSL / TLS
 and privilege separation, and gets this right.

Comments on this document are welcomed.
vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'd() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

Presenting vsftpd's secure design
=================================

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
Separation of responsibilities

TCB: KISS

TCB: Privilege separation

Principle of least privilege

Kerkhoff's principle!

Presenting vsftpd's secure design

vsftpd employs a secure design. The UNIX facilities outlined above are used to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is done in a process running as an unprivileged user. Furthermore, this process runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged child over a socket. All requests are distrusted. Here are example requests:
   - Login request. The child sends username and password. Only if the details are correct does the privileged parent launch a new child with the appropriate user credentials.
   - chown() request. The child may request a recently uploaded file gets chown'ed() to root for security purposes. The parent is careful to only allow chown() to root, and only from files owned by the ftp user.
   - Get privileged socket request. The ftp protocol says we are supposed to emit data connections from port 20. This requires privilege. The privileged parent process creates the privileged socket and passes it to child over the socket.

4) This same privileged parent process makes use of capabilities and chroot(), to run with the least privilege required. After login, depending on what options have been selected, the privileged parent dynamically calculates what privileges it requires. In some cases, this amounts to no privilege, and the privileged parent just exits, leaving no part of vsftpd running with privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSLL protocol parsing is performed in a chroot() jail, running under an unprivileged user. This means both pre-authenticated and post-authenticated OpenSSL protocol parsing; it's actually quite hard to do, but vsftpd manages it in the name of being secure. I'm unaware of any other FTP server which supports both SSL / TLS and privilege separation, and gets this right.

Comments on this document are welcomed.
CHROMIUM ARCHITECTURE

Rendering Engine:
Interprets and executes web content
Outputs rendered bitmaps
The website is the “untrusted code”

Browser Kernel:
Stores data (cookies, history, clipboard)
Performs all network operations

Goal: Enforce a narrow interface between the two
CHROMIUM’S SANDBOX

Makes extensive use of the underlying OS’s primitives

1. Restricted security token
   
   The OS then provides complete mediation on access to “securable objects”
   
   (Security token set s.t. it fails almost always)

2. Separate desktop
   
   Avoid Windows API’s lax security checks

3. Windows Job Object
   
   Can’t fork processes; can’t access clipboard
Goal: Do not leak the ability to read or write the user’s file system

1. Restrict rendering
   Rendering engine doesn’t get a window handle
   Instead, draws to an off-screen bitmap
   Browser kernel copies this bitmap to the screen

2. Network & I/O
   Rendering engine requests uploads, downloads, and file access thru BKI

3. Restrict user input
   Rendering engine doesn’t get user input directly
   Instead, browser kernel delivers it via BKI